Modular Scheduling of Guarded Atomic Actions

Daniel L. Rosenband and Arvind
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{danlief, arvind}@csail.mit.edu

ABSTRACT

A modular synthesis flow is essential for a scalable and hierarchical design methodology. This paper considers a particular modular flow where each module has interface methods and the internal behavior of the module is described in terms of a set of guarded atomic actions on the state elements of the module. A module can also read and update the state of other modules but only by invoking the interface methods of those modules. This paper extends the past work on hardware synthesis of a set of guarded atomic actions by Hoe and Arvind to modules of such actions. It presents an algorithm that, given the *scheduling constraints* on the interface methods of the called modules, derives the "glue logic" and the scheduling constraints for the interface methods of the calling module such that the atomicity of the guarded actions is preserved across module boundaries. Such modules provide reusable IP which facilitates “correctness by construction” design methodology. It also reduces compile-times dramatically in comparison to the compilation that flattens all the modules first.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids – automatic synthesis, hardware description languages.

General Terms

Algorithms, Design, Languages, Verification.

1. ATOMIC ACTIONS: A BASIS FOR HDL

There has been a strong interest in high-level design languages which can bridge the gap between behavioral modeling and efficient hardware synthesis. Commercial efforts have focused either on raising the level of RTL languages so that they are more suitable for modeling (e.g., Verilog to Behavioral Verilog) or on finding suitable extensions and restrictions on conventional languages so that they are more appropriate as an HDL (e.g., C or C++ to SystemC). Typically, Control Data Flow Graphs (CDFG’s) are extracted from the source language program and techniques for compiling SIMD and VLIW architectures are used to generate register transfer logic[8, 9]. These efforts have yet to provide a language that is widely accepted for hardware synthesis. Another type of research has focused on synthesis of specialized versions of programmable processors [10, 15]. These efforts are only tangentially related to general purpose HDLs because the primary focus is on processor issues such as instruction encodings and the automatic generation of assemblers, compilers, etc.

In the research community two other types of languages have been explored that have the potential to raise the level of HDL's. One type of effort is based on synchronous specification languages such as Esterel, Signal, and Lustre which were all designed to deal with real-time issues[3]. Berry[4] and Edwards[7] have presented methods to generate hardware from Esterel but these efforts have yet to yield high quality hardware in comparison to synthesis from Verilog RTL. The other effort is based on asynchronous languages employing atomic actions, which have been used for decades to describe distributed algorithms[5, 13]. Some of the examples in the hardware domain are Dill's Murphi[6], Straunstrup's Synchronous transactions[16], Sere's Action systems[14], and Arvind & Shen’s TRS’s[1]. The main idea underlying all such behavioral descriptions is that any hardware system has a (structural) state component that can be captured by a set of variables that represent registers or storage, and the behavior is nothing but a set of rules, i.e. atomic actions with guards, on this state. A precise and useful semantics emerges from the fact that any legitimate behavior of the system can be understood as a series of atomic actions on this state.

Hoe and Arvind have shown that such atomic descriptions are also amenable to efficient hardware synthesis if the actions of a rule are assumed to take effect in one clock cycle[11, 12]. Hoe's compiler had three types of built-in state elements, i.e., registers, arrays and FIFO's, and accepted as input a set of rules (guarded atomic actions) on these state elements. It then performed analysis based on "rule conflicts" to produce the update logic for state elements as well as a hardware scheduler that allowed many of the enabled rules to execute in parallel. The compiler generated RTL Verilog which could be further synthesized to ASIC's or FPGA's using standard commercial synthesis tools.

This work provided the basis for the development of Bluespec[2], an object oriented HDL. In Bluespec an object represents a hardware module with internal state, rules to manipulate the state, and an interface (a set of methods) through which other modules can observe and manipulate this state. In contrast to Hoe’s compiler, Bluespec has the power to express FIFO’s, arrays and any other hardware building-block as a user defined module using only registers. However, the current Bluespec compilation process is not modular because it flattens (i.e., merges) each non-primitive module before using Hoe’s analysis to generate RTL Verilog.

The initial motivation for introducing a modular compilation flow grew out of our work on microprocessor synthesis where we used FIFO’s parameterized by recursive search functions. The search functions returned values that could be bypassed to earlier pipeline stages. These descriptions were more parameterized than would be required for a specific processor implementation but our expectation was that with proper synthesis algorithms the final circuit would be equivalent to a hand-coded RTL implementation. If successful, this highly-parameterized FIFO could be used across many different designs. Unfortunately, without a modular flow, synthesis times for the processor descriptions were excessive and scheduling results were unsatisfactory because the desired amount of concurrency was not achieved. We address both of these issues in the modular flow described in this paper.

The biggest impact of modular compilation is that a designer can build highly-parameterized modules whose interface scheduling or concurrency properties are not left to the vagaries of the compiler. For example, we can build a FIFO which permits concurrent enqueue and dequeue operations on it, or we can build a register file with two concurrent write ports where the outside logic guarantees that the two write addresses are distinct. Such a FIFO or a register file can then be used in larger designs without having to worry about concurrency issues relating to its interface methods. The current Bluespec compilation scheme does not provide any such assurance except for primitive modules. Modular compilation is also significantly faster than the flat flow.

Paper Organization: In Section 2, we introduce atomic actions and present a synthesis approach that generates efficient hardware from a set of atomic actions. Section 3 extends the ideas from section 2 to produce a modular synthesis algorithm. Section 4 provides experimental results that illustrate some of the benefits of the modular flow. We conclude in section 5.

2. SYNTHESIS OF ATOMIC ACTIONS

This section reviews the execution model of atomic actions and outlines the synthesis approach of Hoe[11, 12].

2.1 Atomic Action Execution Model

Each atomic action (or rule) consists of a body and a guard. The body describes the execution behavior of the rule if it is enabled. The guard (or predicate) specifies the condition that needs to be satisfied for the rule to be executable. We write rules in the form:

rule R_i: when $\pi_i(s)$ ==> $s := \delta_i(s)$

Here, π_i is the predicate and $s := \delta_i(s)$ is the body of rule R_i. Function δ_i is used to compute the next state of the system from the current state s. The execution model for a set of rules is to non-deterministically pick a rule whose predicate is true and then to atomically execute that rule's body. The execution continues as long as some predicate is true:

```
while (some $\pi$ is true) do
    1)  select any  $R_i$ , such that $\pi_i(s)$ is true
    2)  $s := \delta_i(s)$
```

2.2 Synthesizing Rules into RTL Hardware

There is a straightforward translation from rules into hardware. Assuming all state is accessible (no port contention), each π and δ can be implemented easily as combinational logic. A hardware scheduler and control circuit then needs to be added so that in every cycle the scheduler dynamically picks one δ function whose corresponding π condition is satisfied and the control circuit updates the state of the system with the result of the selected δ function. The cycle time in such a synthesis is determined by the slowest π and the slowest δ functions.

Although correct, such an implementation has unsatisfactory performance because it is often possible to execute several rules simultaneously such that the result of the execution matches an execution in which the selected rules are applied in some sequential order. Thus, the challenge in generating efficient hardware from sets of atomic actions is to generate a scheduler which in every cycle picks a maximal set of rules that can be executed simultaneously. In this paper we assume that each rule executes within a single cycle, but we are also investigating implementations where the execution of a rule may stretch over multiple cycles.

Both Staunstrup[16] and Hoe[11, 12] made the observation that two rules can execute simultaneously if they are "conflict free", that is, they do not update the same state and neither updates the state accessed (i.e., "read") by the other rule. Arvind and Hoe further observed that two rules (R_1 and R_2) can execute simultaneously if one rule (R_2) does not read any of the state that the other rule (R_1) writes. In this case, simultaneous execution of R_1 and R_2 appears the same as sequential execution of R_1 followed by R_2. For this to hold, R_2 writes must take precedence over writes to the same state by R_1 and the execution of R_1 must not disable R_2. Such rules are called "sequentially composable" in[12]. Hoe showed that from these pair-wise relationships between rules one can deduce if a group of rules can be scheduled concurrently. Figure 1 shows the circuit that is generated in Hoe's synthesis flow. The predicates (π_i's) are computed for each rule using a combinational circuit. The scheduler is designed to select a maximal subset of applicable rules with the constraint that the outcome of a scheduling step can be explained as atomic firing of rules in some sequence. Based on which rules the scheduler chooses to enable (φ_i's), the selector block then combines the update functions (δ_i's) from the chosen rules and updates the current state with the resulting values.

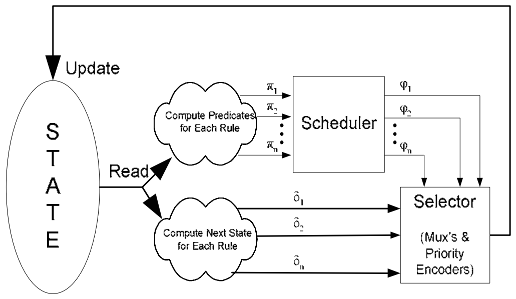

Figure 1: Synthesized Atomic Actions

An aggressive "mutual exclusion" analysis of rules is required to eliminate scheduling cases that cannot arise logically. Without such an analysis one may unnecessarily commit resources, such as ports. One also needs a policy for selecting among the maximal schedules because different maximal sets can have different resource requirements. Construction of good schedulers is the most important problem in synthesis of atomic actions.

It is useful to contrast the synthesis from atomic actions with the behavioral synthesis from CDFG's. As we have seen, the synthesis from atomic actions generates a dynamic scheduler to re-evaluate (in every cycle) which of the enabled rules should be executed concurrently. In contrast, compilation of CDFG's focuses on generating an efficient static schedule of operations over a sequence of control steps. This in turn affects and is affected by how physical resources are allocated to each of these operations. We believe dynamic scheduling is important in hardware systems because many designs have 1. a large number of data dependant conditional paths, each with its own timing and resource requirements, 2. subsystems with variable and unpredictable latencies (due to caching and interference from other processes, etc.), and 3. input events whose timing is often unpredictable. If a hardware system has none of these properties, synthesis from CDFG's should work fine in principle. Otherwise, one has to make many conservative assumptions about timing and conditional paths to generate circuits from CDFG's and such circuits tend to be suboptimal.

3. MODULAR COMPILATION

In this section we first outline the basic semantics of modules and then discuss interface annotations for scheduling. We then present an algorithm to generate the glue logic inside a module that calls other modules. Finally, we show how scheduling annotations can be derived for the methods of a module.

3.1 Modules

Each module in Bluespec contains local state (i.e, instances of primitive modules such as registers), local rules, and interface methods that can be called by other modules. Methods, like rules, contain a body that accesses primitive state elements and call methods of other modules. Each method also contains a guard that indicates to the caller that this method should not be invoked unless the guard is true. For example, the dequeue method in a FIFO has a guard to indicate that the FIFO is not empty.

In the *flat* compilation flow all modules are merged to form a single module. Our *modular* flow is deemed to be correct if it produces the same functional behavior as the completely flattened design. Flattening of modules may be understood in terms of the following procedure which merges two arbitrary modules m_1 and m_2 to produce a new module m. Assume that all module names, primitive state elements and method names are pair-wise unique:

```
FLATTEN($m_1$ , $m_2$) =
  1. define a new module m such that
       m.state   = $m_1$.state $\cup$ $m_2$.state;
       m.rules   = $m_1$.rules $\cup$ $m_2$.rules;
       m.methods  = $m_1$.methods $\cup$ $m_2$.methods;
  2. **foreach** method call $m_i$.h in m where $m_i \in \{m_1, m_2\}$
       - inline the body of $m_i$.h
       - conjugate the guard of the method $m_i$.h to the
         predicate of the rule or the method that calls $m_i$.h
  3. Substitute module name m for all uses of module
     names $m_1$ and $m_2$ in other modules.
```

Notice, step 2 terminates as long as the method calls of m_1 and m_2 do not form a cycle. Also, after merging we can erase all methods (except for those of the root module) which are no longer called from other modules.

To better understand module flattening consider the following example. Suppose we have rule *R: when p ==> m.g(a)* where method *g* inside module *m* in turn invokes methods $m_1.h_1$ with argument $f_1(a)$ and $m_2.h_2$ with argument $f_2(a)$ and is guarded by predicate p_g. After flattening, the compiler turns rule *R* into: *when p &* p_g *==>* $m_1.h_1(f_1(a)); m_2.h_2(f_2(a))$. This indicates that the rule must not fire unless p_g and *p* are both true. Hence, we expose the guard in the modular flow in the form of a ready signal. Similarly, if the rule executes, all the methods that a rule calls must in turn execute all the methods that they call. This is signaled through the enable wire which must always be asserted when the rule executes. Hence, each "action" method of a module has two control signals: enable (en) and ready (rdy), while a "read" method has only a ready control signal (see Figure 2). For correctness, an enable signal should not be asserted unless the corresponding ready signal is true.

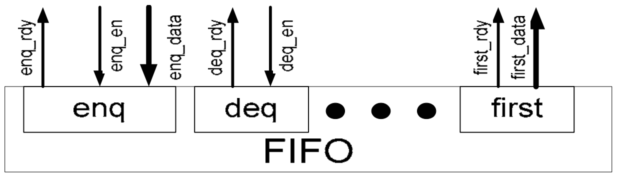

Figure 2: FIFO Interface

A part of the modular compilation problem is how to generate a scheduler and control circuit for a set of rules given only the interface information about the modules whose methods these rules call. For atomicity, if a rule invokes several action methods then *all* those methods must be enabled whenever the rule is elected for execution. Furthermore, if several rules are scheduled for execution then all the methods invoked by them collectively must be enabled. There are several reasons why this may not be allowed by the scheduler. First, there may be resource conflicts – in general, only one call can be made to any method, especially if it has a parameter. Second, the scheduler needs to ensure that the appearance of sequential and atomic execution among rules is maintained. If a rule calls several methods, then their execution must appear atomic with respect to all other rules that are executing simultaneously, even if the called methods are from the same module and may access shared state. Thus, the problem in creating an efficient modular flow is deciding which rules can execute simultaneously while preserving the atomic execution property across module boundaries. The next subsection outlines the scheduling annotations that need to be provided for each pair of methods of a module for correct scheduling.

3.2 Scheduling Annotations

Scheduling annotations describe the effect of an action method (g_1) on the other read and action methods (g_2) of the same module. If two methods are *mutually exclusive* (ME) they obviously cannot affect each other since they will never be called simultaneously. We also assume that the methods of two different modules do not affect each other. (This is true so as long as the method call graph forms a tree of modules. If the call graph is not a tree we can make it so by merging selective modules using the FLATTEN procedure given in the previous section). Annotations must specify 1. if g_1 and g_2 can be called from a single rule; 2. if g1 and g2 are called from different rules can they be scheduled in parallel, and if so, then do they impose any ordering on those rules; and 3. if g_1 can be called from two different rules simultaneously. We do not permit the same action method to be called more than once from a single rule. Read methods on the other hand do not interfere with each other and can be called multiple times from within a rule. (The only complication for read methods is when they have arguments and then the read port becomes a resource to be allocated).

Table 1 shows the scheduling annotations that we use in the modular compilation flow. To understand this table one must keep in mind that when several methods are invoked from a rule, hardware will perform all the reads first and then perform all the state updates simultaneously, all in the same cycle. We refer to this as parallel composition and represent it as $g_1 \oplus g_2$. The meaning of $g_1 < g_2$ is that effects of both g_1 and g_2 are observable but in case of any shared state the updates of g_2 take precedence ("sequential composition" discussed earlier). An "X" indicates that no valid behavior will be observed. Since a module does not know if two methods are being called from a single rule or from two rules, the single rule and two rule behaviors must clearly be equivalent if both are valid. We record these annotations for each pair of methods of a module in a *Conflict Matrix* (CM).

Table 1: Interface Method Annotations

Annotation	1-Rule Behavior	2-Rule Behavior	Example
ME	don't care	don't care	$g_1 = e_1$ when (x == 0) $g_2 = e_2$ when (x == 1)
CF	$g_1 \oplus g_2$	$g_1 < g_2$ $\equiv g_2 < g_1$	g_1 = x := 5 g_2 = y := 6
<	$g_1 \oplus g_2$	$g_1 < g_2$	g_1 = x := y g_2 = y := 5
>	$g_1 \oplus g_2$	$g_2 < g_1$	g_1 = x := 5 g_2 = y := x
P	$g_1 \oplus g_2$	X	g_1 = x := y g_2 = y := x
$<_R$, $>_R$ / EXT	X	$g_1 < g_2 \neq$ $g_2 < g_1$	g_1 = x := 5 g_2 = x := 6
$<_R$	X	$g_1 < g_2$	g_1 = x := x+1 g_2 = x := 6
$>_R$	X	$g_2 < g_1$	g_1 = x := 6 g_2 = x := x+1
C	X	X	g_1 = x := x+1 g_2 = x := x+1

Several things are worth noting about the annotations. The P annotation says that the parallel behavior of g_1 and g_2 ($g_1 \oplus g_2$) is not explainable as sequential behavior of g_1 and g_2. Hence, two rules containing such method calls cannot be scheduled simultaneously. Even though annotation ($<_R$, $>_R$) makes sense we do not allow it for pragmatic reasons. It would require the scheduler to pass the information into the module about what order it has chosen for g_1 and g_2. We require the module to make this choice and specify it in its CM as $<_R$ or as $>_R$.

An action method is not allowed to be invoked more than once from two different rules. However, there is one interesting case which corresponds to the annotation EXT. Consider the action method "g(a) = x := a". Suppose one rule calls "g(3)" and another rules calls "g(4)". It is possible to wire the module externally so that either argument 3 or 4 is passed to g and allow both rules to be scheduled concurrently. We indicate this property of an action method with the annotation EXT. EXT can only occur as a diagonal entry in a CM. As an example of a module's annotation, Table 2 shows the annotations for the primitive register element.

Table 2: Register Annotations

g_1 \ g_2	read	write
read	CF	<
write	>	EXT

3.3 Rule Scheduling Using Module Interface Annotations

Before generating circuits and schedulers for a set of rules, we need to verify that each rule is valid, i.e., it does not attempt to modify the same state more than once. Hence, as long as all pairs of methods in the rule have valid parallel (single rule) execution behavior, the rule is valid. Thus, if $CM[g_1][g_2] \in \{C, <_R, >_R, ME\}$ for pairs of calls in a rule then the rule is not valid. Technically ME is not invalid, but since it implies that the rule will never execute, we flag it as an error. We apply the following VALIDRULE? procedure to each rule to determine its validity. *ActionCalls*(R) returns the set of action method calls made by rule R.

```
VALIDRULE?(R) =
  foreach  $m_i.g_a \in$ ActionCalls(R)
    foreach  $m_j.g_b \in$ (ActionCalls(R) - $m_i.g_a$)
      if (($m_i$ == $m_j$) & ($CM_{m_i}[g_a][g_b] \in \{C, <_R, >_R, ME\}$)) then
        return FALSE;
    return TRUE;
```

Next, we need to determine if each pair of rules R_1 and R_2 can be scheduled simultaneously. Suppose we want to know if it will appear as though R_1 executes before R_2. For this to hold, it must be true that it will appear that every method that is called in R_1 will occur before every method in R_2. Thus, we start with the assumption that such scheduling is possible, and constrain the result as we examine each pair of method calls:

```
DERIVEREL($R_1$, $R_2$) =
  result = CF;
  foreach  $m_i.g_a \in$ Calls($R_1$)
    foreach  $m_j.g_b \in$ Calls($R_2$)
      if ($m_i$ == $m_j$) then
        if ($CM_{m_i}[g_a][g_b]$ == ME) then
          return ME;
        else
          result = LUB(result, $CM_{m_i}[g_a][g_b]$);
  return result;
```

The least-upper-bound (LUB) operator in this procedure is defined over the lattice of annotations in Figure 3. The smallest value in this lattice is CF, the largest is *C*.

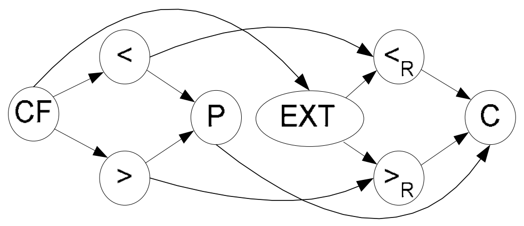

Figure 3: Annotation Lattice

If the result of DERIVEREL(R_1, R_2) is an element of the set {CF, <, EXT, $<_R$}, then enabling R_1 and R_2 simultaneously will appear as though R_1 executes before R_2 (provided of course, the circuits to call the methods that R_1 and R_2 call are correct). Thus, for each pair of rules, we can determine their sequential scheduling relationship. This is precisely the information that Hoe's[11, 12] synthesis algorithm requires to generate a scheduler. Thus, we can derive the pairwise rule information using the procedure above and then feed it directly into Hoe's unmodified scheduler. Note: This works when we are just compiling rules. We will see that some modifications are required when scheduling a module's rules together with the module's methods.

Circuit generation requires us to incorporate the ready signals of the called methods and assert the enable signals and supply input parameters for all the called methods. The outputs (results) of method calls can be fed directly into combinational logic. The circuits take the following form:

- $\pi_{i\_new} = \pi_{i\_old}$ & $m_x.g_a$.rdy & $m_y.g_b$.rdy & …
  where $m_x.g_a$.rdy & $m_y.g_b$.rdy & … is the conjunction of all ready signals of the methods that rule R_i calls.
- $m_i.g_a$.en = $\varphi_x \mid \varphi_y \mid$ …
  where $\varphi_x \mid \varphi_y \mid$ … is the disjunction of all φ's of rules that call method $m_i.g_a$
-
```
if (DERIVEREL(m_i.g_a, m_i.g_a) == EXT) then
  m_i.g_a.data = parameter value that the last rule that is
                 scheduled and that calls m_i.g_a contains.
else
  m_i.g_a.data = (φ_x & (R_x's parameter to m_i.g_a)) |
                 (φ_y & (R_y's parameter to m_i.g_a)) | …
   where R_x, R_y, … are the rules that call method m_i.g_a
```

The use of the ready signals and generation of the enable signals is straightforward. Input parameter value generation depends on the type of method being called. If the method has an EXT annotation, then the rule that appears to execute after all other rules in the schedule passes the value to the method. Assuming a fixed relative scheduling ordering among rules, this can be implemented as a priority encoder. A multiplexor can be used for all non-EXT methods.

It should be noted that method interfaces (ports) can be viewed as resources in this scheduling / circuit generation approach. The same method cannot be called twice in the same cycle except for the EXT case. Another exception occurs when a purely combinational method is called with the same arguments in two rules. In this case, both rules can share the result of the return value.

3.4 Deriving Module Interface Annotations

The same procedure that was used to derive the scheduling relationship among rules (DERIVEREL) can be used to determine the scheduling relationship (interface annotation) of interface methods: DERIVEREL(g_1, g_2) returns the interface annotation for methods g_1 and g_2. As with rules, we also need to perform a validity check on every method to ensure that it does not invoke a pair of methods that update the same state. We also need to select between $<_r$ or $>_r$ in case a pair of methods can be scheduled in either order since we do not permit the caller to chose the order dynamically.

Sometimes the derived annotations are more restrictive than what a designer had expected. This is usually due to the fact that the designer has information which either a compiler does not have or can not derive based on the procedures given here. We first encountered this issue while designing processor pipelines using FIFO's. Our implementation allowed simultaneous enqueue and dequeue operations but the compiler could not deduce that an enqueue can never disable a dequeue. Even a tougher case was encountered while designing the reorder buffer (rob) of a microprocessor. Higher level logic ensured that the two simultaneous writes into the rob could never be to the same slot but this fact is not deducible from the rule analysis without a theorem prover. In all such cases we found that the best solution was to allow the designer to overrule the annotations deduced by the compiler. This solution limits the scope of the design that needs to be examined for verification. In fact we can use the FIFO with our less strict annotations as part of the library without causing any problems for the users of the FIFO module.

3.5 Module Compilation

An important observation when compiling rules together with a module's interface methods is that interface methods are nearly identical to rules. The only difference is in the way they are scheduled. Rule scheduling is a local operation within a module. In contrast, methods are scheduled external to the module. Whether or not the method executes is indicated through the enable signal. Thus, the enable signal can be thought of as the method's φ signal.

Surprisingly, the module interface annotations have implications for rule scheduling inside the module as well. In general a module does not know if two methods are being invoked from one rule or from two rules. The semantics must be such that the module behaves correctly in either case, provided the external scheduler is following the constraints imposed by the interface. When two methods are called from one rule then it must appear as if the external rule (together with the methods it calls) executes atomically with respect to the rules inside the module. Thus, if we do not know if the enabled methods are being called from a single or from multiple rules (because both would be valid executions), then the scheduler must assume that they are being called from a single rule and schedule all internal rules to either occur before or after the methods.

To complete the modular compilation flow we provide the COMPILE procedure:

```
COMPILE(m) =
  1. Compile each module invoked by m (bottom-up)
     foreach module m_i invoked by ,
        COMPILE(m_i);
  2. Compile the  module m
     foreach RorM_a ∈  rules and interface methods of m
       if (!VALIDRULE?(RorM_a)) then
          return ERROR!!!
       foreach RorM_b ∈ rules and interfaces methods of m
         CM_s[RorM_a][RorM_b] = DERIVEREL(RorM_a,RorM_b)
  3. GENERATESCHEDULER
  4. GENERATECIRCUIT
```

This procedure performs a bottom-up compile. After compiling all child modules, it uses the child module annotations to generate the scheduler and circuits for the parent module's rules and methods. The GENERATESCHEDULER procedure is equivalent to the one described in section 3.3, with the slight modifications outlined in this section. The GENERATECIRCUIT procedure is equivalent to the circuit generation described in section 3.3 since the interface method circuits are generated in exactly the same way as rule circuits.

4. RESULTS

Table 3 summarizes scheduling and compile time results from experimentation on several processor models. These examples illustrate the dramatic improvement in compile times that we see when using the modular flow. They also show that scheduling improves over the flat approach if we allow the designer to alter scheduling at some of the interfaces.

We worked with two ISA's, one very simple design that contains 5 instructions (5I) and one that implements a MIPS-II core. The MIPS core is implemented as a fully bypassed 5-stage pipeline. In order to stress the synthesis, all designs used a complex, recursive definition of a highly-parameterized FIFO as pipeline / bypass registers. The only primitive module that was used in all designs was the primitive register. Simulations of binaries running on each processor were used to verify their functionality.

Each processor was synthesized using both the flat Bluespec flow and the modular flow. The modular flow compiled the FIFO, and in one case also the register file (RF), as a separate module. Because of the complexity of the FIFO description, the compiler could not derive optimal annotations in the modular flow, or rule schedules in the flat flow. However, by allowing the designer to alter the annotations of the FIFO module (something he only has to do once and can then reuse in all processor designs), we were able to achieve optimal schedules in all modular compilations.

Table 3: Flat vs. Modular Compilation

Processor	Optimal Schedule	Partial Eval.	Scheduler	Total
5I 2-Stage Flat	No	0.7s	1.0s	3.2s
5I 2-Stage Modular	Yes	0.1s	0.1s	2.0s
5I 5-Stage Bypass Flat	No	26.8s	Opt. OFF	29.4s
5I 5-Stage Bypass Modular	Yes	0.9s	0.2s	3.6s
MIPS Flat	No	1036s	Opt. OFF	1052s
MIPS Modular FIFO	Yes	46.0s	218.1s	275.8s
MIPS Modular FIFO + RF	Yes	21.9s	1.8s	35.7s

The two largest compilation phases are partial evaluation and scheduling. The partial evaluation phase expands the code by inlining functions and modules, performs partial evaluation wherever possible, unrolls recursive calls, etc. The scheduling phase of the compiler generates the scheduler -- decides which rules are mutually exclusive, conflicting, etc. In both of these phases the modular flow is significantly faster than the flat flow. This is largely due to fewer rules needing to be compiled when using the modular flow and due to the reduction in the size of expressions. In the scheduling phase, not all optimizations could be turned on in the flat flow because expression sizes got too large for analysis that is exponential in its runtime. As expected, the total compile time is dramatically less in the modular flow. We should note that area and timing were nearly identical in the two compilation approaches and closely matched results from a hand-coded implementation.

5. CONCLUSION

In this paper we presented an algorithm for modular compilation of atomic actions. This compilation strategy greatly improves compile times, which in turn makes experimentation with larger designs more practical. Through the use of scheduling annotations at module boundaries we were also able to build libraries using the full power of the Bluespec language, without relaxing requirements on scheduling efficiency. The combination of compile time improvements and ability to build libraries native to the Bluespec language fundamentally strengthens the usability of the infrastructure. We can now take full advantage of the language, build libraries with significant intellectual property using it, and still get the performance we expect.

6. REFERENCES

[1] Arvind and Shen, X. Using term rewriting systems to design and verify processors. *Micro, IEEE*, *19* (3). 36-46.

[2] Augustsson, L. and others. Bluespec: Language definition, Sandburst Corp., 2001.

[3] Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P. and de Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE*, *91* (1). 64-83.

[4] Berry, G. Esterel on hardware. *Philos. Trans. Roy. Soc. London* (Series A, 339). 87-104.

[5] Chandy, K.M. and Misra, J. *Parallel program design : a foundation*. Addison-Wesley Pub. Co., Reading, MA, 1988.

[6] Dill, D.L. The Murphi verification system. in *Proceedings of the Eigth International Conference on Computer-Aided Verification*, Springer-Verlag, 1996.

[7] Edwards, S.A., High-level Synthesis from the Synchronous Language Esterel. in *Proceedings of the International Workshop of Logic and Synthesis (IWLS)*, (New Orleans, Louisiana, 2002).

[8] Gajski, D.D. *High-level synthesis : introduction to chip and system design*. Kluwer Academic, Boston, 1992.

[9] Gupta, S., Dutt, N.D., Gupta, R.K. and Nicolau, A., SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. in *International Conference on VLSI Design*, (2003).

[10] Hadjiyiannis, G., Hanono, S. and Devadas, S., ISDL: An Instruction Set Description Language For Retargetability. in *Proceedings of the 34th Design Automation Conference (DAC)*, (1997), 299-302.

[11] Hoe, J.C. Operation-centric hardware description and synthesis *Dept. of Electrical Engineering and Computer Science*, Massachusetts Institute of Technology, 2000, 139 p.

[12] Hoe, J.C. and Arvind, Synthesis of operation-centric hardware descriptions. in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, (2000), 511-518.

[13] Lamport, L. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.*, 5 (2). 190-222.

[14] Plosila, J. and Sere, K., Action systems in pipelined processor design. in Proceedings Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, (1997), 156-166.

[15] Schliebusch, O., Hoffmann, A., Nohl, A., Braun, G. and Meyr, H., Architecture implementation using the machine description language LISA. in Proceedings 7th Asia and South Pacific Design Automation Conference (ASP-DAC), (2002), 239-244.

[16] Staunstrup, J. and Greenstreet, M.R. From High-Level Descriptions to VLSI Circuits. BIT, 28 (3). 620-638.